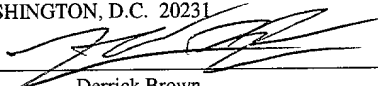


PATENT
5500-65900
TT4235

"EXPRESS MAIL" MAILING LABEL NUMBER
EL726370563US

DATE OF DEPOSIT APRIL 2, 2001

I HEREBY CERTIFY THAT THIS PAPER OR
FEE IS BEING DEPOSITED WITH THE
UNITED STATES POSTAL SERVICE
"EXPRESS MAIL POST OFFICE TO
ADDRESSEE" SERVICE UNDER 37 C.F.R.
§1.10 ON THE DATE INDICATED ABOVE
AND IS ADDRESSED TO THE ASSISTANT
COMMISSIONER FOR PATENTS,
WASHINGTON, D.C. 20231


Derrick Brown

Mode Dependent Segment Use with Mode Independent Segment Update

By:

Kevin J. McGrath

BACKGROUND OF THE INVENTION

1. Field of the Invention

5 This invention is related to the field of processors and, more particularly, to segmentation mechanisms in processors.

2. Description of the Related Art

10 The x86 architecture (also known as the IA-32 architecture) has enjoyed widespread acceptance and success in the marketplace. Accordingly, it is advantageous to design processors according to the x86 architecture. Such processors may benefit from the large body of software written to the x86 architecture (since such processors may execute the software and thus computer systems employing the processors may enjoy
15 increased acceptance in the market due to the large amount of available software).

 The x86 architecture includes a segmentation mechanism and a paging mechanism for performing address translation. Segment descriptors in a segment table provide a segment base address which is added to other operands of an instruction to
20 produce a linear address, which may then be translated through a paging mechanism to a physical address. The segmentation mechanism provides for variable sized segments which may be arbitrarily located in memory. Unfortunately, such segments complicate memory management (particularly in a multitasking environment). The paging mechanism translates fixed size pages of virtual (linear) addresses to corresponding fixed
25 sized pages of physical addresses, thus simplifying memory allocation and management. The segment descriptor may also include various access attributes used to control the allowable access to the segment (e.g. read/write) and to indicate the type of information stored in the segment (e.g. code/data/stack). Additionally, the paging mechanism includes access attributes as well.

In modern operating systems, the demand paged memory allocation model supported by the paging mechanism has become popular while the segmentation mechanism may be largely unused. Particularly, a "flat addressing" mode is frequently
5 used in which the segment descriptors are programmed to a zero base address and a maximum limit. In this manner, the segments may be mapped to the entire linear address space, thus having no effect on the linear address. Effectively, only the paging mechanism is used in this mode. However, it is still desirable to provide compatibility with application programs which use segmentation.

10

SUMMARY OF THE INVENTION

A processor is described which includes multiple protected operating modes. In at least one of the protected operating modes, segmentation is disabled or partially disabled.
15 In other words, segment descriptor information may not affect the execution of an instruction specifying the segment register storing the segment descriptor information. In other protected operating modes, segmentation may not be disabled (e.g. segment descriptor information does affect the execution of the instruction specifying the segment register storing the segment descriptor information). However, segment load instructions
20 may be performed in any protected operating mode (including checking segment descriptors for exception conditions during the segment load), which may allow a code sequence executing in one of the protected operating modes in which segmentation is disabled to establish segments in the segment registers for use by other code sequences executing in other protected operating modes. Thus, for example, an operating system
25 operating in a protected mode in which segmentation is disabled may establish segments for an application program which uses segmentation. Compatibility with programs employing segmentation may be preserved while allowing for protected operating modes in which segmentation is not used.

Broadly speaking, a processor is contemplated including a segment register and an execution core coupled thereto. The segment register is configured to store a segment selector locating a segment descriptor. The execution core is configured to: (i) execute a first instruction specifying the segment register and (ii) update the segment register in response to a segment load instruction independent of which of a plurality of protected operating modes is active. The execution core is selectively responsive to the segment descriptor during execution of the first instruction dependent on which of the plurality of protected operating modes is active in the processor.

10 Additionally, an apparatus is contemplated comprising a storage location and a processor. The storage location corresponds to a segment register and stores a segment selector locating a segment descriptor. The processor is configured to: (i) process a first instruction specifying the segment register, and (ii) update the storage location in response to a segment load instruction independent of which of a plurality of protected operating modes is active. The processing of the first instruction includes selectively responding to the segment descriptor dependent on which of the plurality of protected operating modes is active.

20 Furthermore, a method is contemplated. A first instruction specifying a segment register is executed, the executing including selectively responding to a segment descriptor indicated by a segment selector in the segment register. The selectively responding is dependent on which of a plurality of protected operating modes is active. The segment register in response to a segment load instruction independent of which of the plurality of protected operating modes is active.

25 Still further, a method is contemplated. One or more segment descriptors are loaded into segment registers, the loading performed in a first protected operating mode in which at least one of the one or more segment descriptors is not used even if an instruction specifying the corresponding segment register is executed. A code segment is

branched to, the code sequence establishing a second protected operating mode in which the one or more segment descriptors are used.

BRIEF DESCRIPTION OF THE DRAWINGS

5

The following detailed description makes reference to the accompanying drawings, which are now briefly described.

Fig. 1 is a block diagram of one embodiment of a processor.

10

Fig. 2 is a block diagram of one embodiment of a segment descriptor for 32/64 mode.

15

Fig. 3 is a block diagram of one embodiment of a segment descriptor for compatibility mode.

20

Fig. 4 is a table illustrating one embodiment of operating modes as a function of segment descriptor and control register values.

Fig. 5 is a flowchart illustrating operation of one embodiment of an execution core shown in Fig. 1 during execution of an instruction.

Fig. 6 is a flowchart illustrating operation of one embodiment of an execution core shown in Fig. 1 during execution of a segment load instruction.

25

Fig. 7 is a table illustrating segment descriptor checks according to one embodiment of the processor shown in Fig. 1.

Fig. 8 is a flowchart illustrating one embodiment of an operating system routine

launching an application program.

Fig. 9 is a flowchart illustrating one embodiment of an interpreter.

5 Fig. 10 is a flowchart illustrating one embodiment of a translator.

Fig. 11 is a block diagram illustrating one embodiment of mapping non-native architected state.

10 Fig. 12 is a block diagram illustrating a second embodiment of mapping non-native architected state.

Fig. 13 is a block diagram illustrating a third embodiment of mapping non-native architected state.

15 Fig. 14 is a block diagram of one embodiment of a carrier medium.

Fig. 15 is a block diagram of one embodiment of a computer system including the processor shown in Fig. 1.

20 Fig. 16 is a block diagram of another embodiment of a computer system including the processor shown in Fig. 1.

While the invention is susceptible to various modifications and alternative forms, 25 specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the

appended claims.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

5 **Processor Overview**

Turning now to Fig. 1, a block diagram illustrating one embodiment of a processor 10 is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 1, processor 10 includes an instruction cache 12, an execution core 14, a data cache 16, an external interface unit 18, a memory management unit (MMU) 20, and a register file 22. In the illustrated embodiment, MMU 20 includes a set of segment registers 24, a first control register 26, a second control register 28, a local descriptor table register (LDTR) 30, a global descriptor table register (GDTR) 32, and a page table base address register (CR3) 34. Instruction cache 12 is coupled to external interface unit 18, execution core 14, and MMU 20. Execution core 14 is further coupled to MMU 20, register file 22, and data cache 16. Data cache 16 is further coupled to MMU 20 and external interface unit 18. External interface unit 18 is further coupled to MMU 20 and to an external interface.

Processor 10 may employ a processor architecture compatible with the x86 architecture (also known as the IA-32 architecture) and including additional architectural features to support 64 bit processing. More particularly, the processor architecture employed by processor 10 may define a mode, referred to below as "long mode". Long mode is a mode in which 64 bit processing is selectable as an operating mode, as well as 32 bit or 16 bit processing as specified in the x86 architecture. More particularly, long mode may provide for an operating mode in which virtual addresses may be greater than 32 bits in size.

Processor 10 may implement a mechanism allowing for orderly transition to and from long mode, even though multiple registers may be changed to perform the transition.

Particularly, processor 10 may employ a long mode active (LMA) indication in a control register (e.g. control register 26 in the present embodiment, although the LMA indication may be stored in any control register, including control registers not storing the LME indication). The processor 10 may use the LMA indication as the indication of whether or not long mode is active (i.e. whether or not the processor is operating in long mode).
5 However, the LMA indication may not be modified directly via an instruction. Instead, an instruction is used to change the state of the LME indication to indicate whether or not long mode is desired. Long mode may be activated (as indicated by the LMA indication) via the combination of enabling paging (as indicated by the PG indication in control
10 register 28 and described in more detail below) and the LME indication indicating that long mode is desired. Viewed in another way, the LME indication may be used to enable the transition to long mode. The LMA indication may indicate whether or not the transition has successfully occurred, and thus indicates whether processor 10 is operating according to the long mode definition or processor 10 is operating according to the legacy
15 definition of the x86 processor architecture.

Processor 10 is configured to establish an operating mode in response to information stored in a code segment descriptor corresponding to the currently executing code and further in response to one or more enable indications stored in one or more
20 control registers. As used herein, an "operating mode" specifies default values for various programmably selectable processor attributes. For example, the operating mode may specify a default operand size and a default address size. The default operand size specifies the number of bits in an operand of an instruction, unless an instruction's encoding overrides the default. The default address size specifies the number of bits in an
25 address of a memory operand of an instruction, unless an instruction's encoding overrides the default. The default address size specifies the size of at least the virtual address of memory operands. As used herein, a "virtual address" is an address generated prior to translation through an address translation mechanism (e.g. a paging mechanism) to a "physical address", which is the address actually used to access a memory. Additionally,

as used herein, a "segment descriptor" is a data structure created by software and used by the processor to define a segment of memory and to further define access control and status for the segment. A "segment descriptor table" is a table in memory storing segment descriptors. Since there is more than one operating mode, the operating mode in effect at
5 any given time may be described as being the "active" operating mode.

In the illustrated embodiment, MMU 20 generates an operating mode and conveys the operating mode to execution core 14. Execution core 14 executes instructions using the operating mode. More particularly, execution core 14 fetches operands having the
10 default operand size from register file 22 or memory (through data cache 16, if the memory operands are cacheable and hit therein, or through external interface unit 18 if the memory operands are noncacheable or miss data cache 16) unless a particular instruction's encoding overrides the default operand size, in which case the overriding operand size is used. Similarly, execution core 14 generates addresses of memory
15 operands, wherein the addresses have the default address size unless a particular instruction's encoding overrides the default address size, in which case the overriding address size is used. In other embodiments, the information used to generate the operating mode may be shadowed locally in the portions of processor 10 which use the operating mode (e.g. execution core 14), and the operating mode may be determined from
20 the local shadow copies.

As mentioned above, MMU 20 generates the operating mode responsive to a code segment descriptor corresponding to the code being executed and further responsive to one or more values in control registers. Information from the code segment descriptor is
25 stored in one of the segment registers 24 (a register referred to as CS, or code segment). Additionally, control register 26 stores an enable indication (LME) which is used to enable transition to long mode and the LMA indication indicating whether or not long mode is active. In long mode, an operating mode in which the default address size is greater than 32 bits ("32/64 mode") as well as certain compatibility modes for the 32 bit

and 16 bit operating modes may be available using the segment descriptor indications. The default operand size may be 32 bits in 32/64 mode, but instructions may override the default 32 bit operand size with a 64 bit operand size when desired. If the LME indication is in an enabled state, then long mode may be activated. If the LME indication is in a disabled state, then long mode may not be activated. In one embodiment, the default address size in 32/64 mode may be implementation-dependent but may be any value up to and including 64 bits. Furthermore, the size of the virtual address may differ in a given implementation from the size of the physical address in that implementation.

10 It is noted that various indications are described herein (e.g. LMA, LME, etc.). Generally, an indication is a value which may be placed into two or more states. Each state may be assigned a meaning. Some of the indications described herein (including some enable indications) may be described as bits. The bit being set may be one state (e.g. the enabled state for enable indications) and the bit being clear may be the other state (e.g. the disabled state for enable indications). However, other encodings are possible, including encodings in which multiple bits are used and encodings in which the enabled state is the clear state and the disabled state is the set state. Accordingly, the remainder of this description may refer to the LME indication in control register 26 as the LME bit, with the enabled state being set and the disabled state being clear. However, other encodings of the LME indication are contemplated, as set forth above. Similarly, the LMA indication may be referred to as the LMA bit, with the set state indicating that long mode is active and the clear state indicating that long mode is inactive. However, other encodings of the LMA indication are contemplated, as set forth above.

25 Segment registers 24 store information from the segment descriptors currently being used by the code being executed by processor 10. As mentioned above, CS is one of segment registers 24 and specifies the code segment of memory. The code segment stores the code being executed. Other segment registers may define various data segments (e.g. a stack data segment defined by the SS segment register, and up to four

data segments defined by the DS, ES, FS, and GS segment registers). Fig. 1 illustrates the contents of an exemplary segment register 24A, including a selector field 24AA and a descriptor field 24AB. Selector field 24AA is loaded with a segment selector to activate a particular segment in response to certain segment load instructions executed by

5 execution core 14. The segment selector locates the segment descriptor in a segment descriptor table in memory. More particularly, processor 10 may employ two segment descriptor tables: a local descriptor table and a global descriptor table. The base address of the local descriptor table is stored in the LDTR 30. Similarly, the base address of the global descriptor table is stored in GDTR 32. A bit within the segment selector (the table

10 indicator bit) selects the descriptor table, and an index within the segment selector is used as an index into the selected table. When an instruction loads a segment selector into one of segment registers 24, MMU 20 reads the corresponding segment descriptor from the selected segment descriptor table and stores information from the segment descriptor into the segment descriptor field (e.g. segment descriptor field 24AB for segment register

15 24A). The information stored in the segment descriptor field may comprise any suitable subset of the segment descriptor, including all of the segment descriptor, if desired. Additionally, other information derived from the segment descriptor or other sources may be stored in the segment descriptor field, if desired. For example, an embodiment may decode the operating mode indications from the code segment descriptor and store the

20 decoded value rather than the original values of the operating mode indications. If an instruction causes CS to be loaded with a segment selector, the code segment may change and thus the operating mode of processor 10 may change.

In one embodiment, only the CS segment register is used in 32/64 mode. The data

25 segment registers are ignored from the standpoint of providing segmentation information. In 16 and 32 bit modes, the code segment and data segments may be active. Furthermore, a second enable indication (PE) in control register 28 may affect the operation of MMU

20. The PE enable indication may be used to enable protected mode, in which segmentation and/or paging address translation mechanisms may be used. If the PE

enable indication is in the disabled state, segmentation and paging mechanisms are disabled and processor 10 is in "real mode" (in which addresses generated by execution core 14 are physical addresses). Similar to the LME indication, the PE indication may be a bit in which the enabled state is the bit being set and the disabled state is the bit being clear. However, other embodiments are contemplated as described above. Generally, a "protected mode" is a mode in which various hardware and/or software mechanisms are employed to provide controlled access to memory.

Control register 28 is further illustrated in Fig. 1 as storing a paging enable indication (PG). The PG indication may indicate whether or not paging is enabled. As mentioned above, the LMA bit is set once paging is enabled and the LME bit is set. As used herein, the term "paging" or "paging address translation" refers to the translation of virtual addresses to physical addresses using mappings stored in a page table structure indicated by the page table base address register 34. A given page mapping maps any virtual address having the same virtual page number to a corresponding physical address in a page of physical memory. The page table is a predefined table of entries stored in memory. Each of the entries store information used to map virtual addresses to physical addresses.

It is noted that MMU 20 may employ additional hardware mechanisms, as desired. For example, MMU 20 may include paging hardware to implement paging address translation from virtual addresses to physical addresses. The paging hardware may include a translation lookaside buffer (TLB) to store page translations.

It is noted that control registers 26 and 28 may be implemented as architected control registers (e.g. control register 26 may be CR4 and control register 28 may be CR0). Alternatively, one or both of the control registers may be implemented as model specific registers to allow for other uses of the architected control registers without interfering with 32/64 mode. Generally, the control registers are each addressable by one

or more instructions defined in the processor architecture, so that the registers may be changed as desired.

Instruction cache 12 is a high speed cache memory for storing instruction bytes.

5 Execution core 14 fetches instructions from instruction cache 12 for execution.

Instruction cache 12 may employ any suitable cache organization, including direct-mapped, set associative, and fully associative configurations. If an instruction fetch misses in instruction cache 12, instruction cache 12 may communicate with external interface unit 18 to fill the missing cache line into instruction cache 12. Additionally,
10 instruction cache 12 may communicate with MMU 20 to receive physical address translations for virtual addresses fetched from instruction cache 12.

Execution core 14 executes the instructions fetched from instruction cache 12.

Execution core 14 fetches register operands from register file 22 and updates destination
15 registers in register file 22. The size of the register operands is controlled by the operating mode and any overrides of the operating mode for a particular instruction.

Similarly, execution core 14 fetches memory operands from data cache 16 and updates destination memory locations in data cache 16, subject to the cacheability of the memory operands and hitting in data cache 16. The size of the memory operands is similarly
20 controlled by the operating mode and any overrides of the operating mode for a particular instruction. Furthermore, the size of the addresses of the memory operands generated by execution core 14 is controlled by the operating mode and any overrides of the operating mode for a particular instruction.

25 Execution core 14 may employ any suitable construction. For example, execution core 14 may be a superpipelined core, a superscalar core, or a combination thereof.

Execution core 14 may employ out of order speculative execution or in order execution, according to design choice. Execution core 14 may include microcoding for one or more instructions or exception situations, in combination with any of the above constructions.

Register file 22 may include 64 bit registers which may be accessed as 64 bit, 32 bit, 16 bit, or 8 bit registers as indicated by the operating mode of processor 10 and any overrides for a particular instruction. The registers included in register file 22 may include the RAX, RBX, RCX, RDX, RDI, RSI, RSP, and RBP registers (which may be 64 bit versions of the EAX, EBX, ECX, EDX, EDI, ESI, ESP, and EBP registers defined in the x86 processor architecture, respectively). Additionally, in one embodiment, register file 22 may include additional registers addressed using a register extension (REX) prefix byte. Register file 22 may further include the RIP register, which may be a 64 bit version of the EIP register. Alternatively, execution core 14 may employ a form of register renaming in which any register within register file 22 may be mapped to an architected register. The number of registers in register file 22 may be implementation dependent for such an embodiment.

Data cache 16 is a high speed cache memory configured to store data. Data cache 16 may employ any suitable cache organization, including direct-mapped, set associative, and fully associative configurations. If a data fetch or update misses in data cache 16, data cache 16 may communicate with external interface unit 18 to fill the missing cache line into data cache 16. Additionally, if data cache 16 employs a writeback caching policy, updated cache lines which are being cast out of data cache 16 may be communicated to external interface unit 18 to be written back to memory. Data cache 16 may communicate with MMU 20 to receive physical address translations for virtual addresses presented to data cache 16.

External interface unit 18 communicates with portions of the system external to processor 10. External interface unit 18 may communicate cache lines for instruction cache 12 and data cache 16 as described above, and may communicate with MMU 20 as well. For example, external interface unit 18 may access the segment descriptor tables and/or paging tables on behalf of MMU 20.

It is noted that processor 10 may include an integrated level 2 (L2) cache, if desired. Furthermore, external interface unit 18 may be configured to communicate with a backside cache in addition to communicating with the system.

5

While the processor architecture described herein may be compatible with the x86 processor architecture for 16 and 32 bit modes, in one embodiment, other embodiments may employ any 16 and 32 bit modes. The other embodiments may or may not be compatible with the x86 processor architecture or any other processor architecture. It is further noted that, while a specific set of information is described herein as being used to generate the operating mode, any combination of indications and/or information from memory data structures such as segment descriptor tables and page tables may be used to generate the operating mode in various embodiments.

Turning now to Fig. 2, a block diagram of one embodiment of a code segment descriptor 40 for 32/64 mode is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 2, code segment descriptor 40 comprises 8 bytes with the most significant 4 bytes illustrated above the least significant 4 bytes. The most significant four bytes are stored at a numerically larger address than the least significant four bytes. The most significant bit of each group of four bytes is illustrated as bit 31 in Fig. 2 (and Fig. 3 below), and the least significant bit is illustrated as bit 0. Short vertical lines within the four bytes delimit each bit, and the long vertical lines delimit a bit but also delimit a field (both in Fig. 2 and in Fig. 3).

Unlike the 32 bit and 16 bit code segment descriptors illustrated in Fig. 3 below, code segment descriptor 40 does not include a base address or limit. Processor 10 employs a flat virtual address space for 32/64 mode (rather than the segmented linear address space employed in 32 bit and 16 bit modes). Accordingly, the portions of code segment descriptor 40 which would otherwise store the base address and limit are

reserved in segment descriptor 40. It is noted that a virtual address provided through segmentation may also be referred to herein as a "linear address". The term "virtual address" encompasses any address which is translated through a translation mechanism to a physical address actually used to address memory, including linear addresses and other
5 virtual addresses generated in non-segmented architectures.

Segment descriptor 40 includes a D bit 42, an L bit 44 (set to one for a 32/64 mode code segment), an available bit (AVL) 46, a present (P) bit 48, a descriptor privilege level (DPL) 50, and a type field 52. D bit 42 and L bit 44 are used to determine
10 the operating mode of processor 10, as illustrated in Fig. 4 below. AVL bit 46 is available for use by system software (e.g. the operating system). P bit 48 is used to indicate whether or not the segment is present in memory. If P bit 48 is set, the segment is present and code may be fetched from the segment. If P bit 48 is clear, the segment is not present and an exception is generated to load the segment into memory (e.g. from disk
15 storage or through a network connection). The DPL indicates the privilege level of the segment. Processor 10 employs four privilege levels (encoded as 0 through 3 in the DPL field, with level 0 being the most privileged level). Certain instructions and processor resources (e.g. configuration and control registers) are only executable or accessible at the more privileged levels, and attempts to execute these instructions or access these
20 resources at the lower privilege levels result in an exception. When information from code segment 40 is loaded into the CS segment register, the DPL becomes the current privilege level (CPL) of processor 10. Type field 52 encodes the type of segment. For code segments, the most significant bit two bits of type field 52 may be set (the most significant bit distinguishing a code or data segment from a system segment, and the
25 second most significant bit distinguishing a code segment from a data segment), and the remaining bits may encode additional segment type information (e.g. execute only, execute and read, or execute and read only, conforming, and whether or not the code segment has been accessed).

It is noted that, while several indications in the code segment descriptor are described as bits, with set and clear values having defined meanings, other embodiments may employ the opposite encodings and may use multiple bits, as desired. Thus, for example, the D bit 42 and the L bit 44 may each be an example of an operating mode indication which may be one or more bits as desired, similar to the discussion of enable indications above.

Turning now to Fig. 3, a block diagram of one embodiment of a code segment descriptor 54 for 32 and 16 bit compatibility mode is shown. Other embodiments are possible and contemplated. As with the embodiment of Fig. 2, code segment descriptor 54 comprises 8 bytes with the most significant 4 bytes illustrated above the least significant 4 bytes.

Code segment descriptor 54 includes D bit 42, L bit 44, AVL bit 46, P bit 48, DPL 50, and type field 52 similar to the above description of code segment descriptor 40. Additionally, code segment descriptor 54 includes a base address field (reference numerals 56A, 56B, and 56C), a limit field (reference numerals 57A and 57B) and a G bit 58. The base address field stores a base address which is added to the logical fetch address (stored in the RIP register) to form the linear address of an instruction, which may then optionally be translated to a physical address through a paging translation mechanism. The limit field stores a segment limit which defines the size of the segment. Attempts to access a byte at a logical address greater than the segment limit are disallowed and cause an exception. G bit 58 determines the scaling of the segment limit field. If G bit 58 is set the limit is scaled to 4K byte pages (e.g. 12 least significant zeros are appended to the limit in the limit field). If G bit 58 is clear, the limit is used as is.

It is noted that code segment descriptors for 32 and 16 bit modes when long mode is not active may be similar to code segment descriptor 54, except the L bit is reserved and defined to be zero. It is further noted that, in 32 and 16 bit modes (both compatibility

mode with the LMA bit set and modes with the LMA bit clear) according to one embodiment, data segments are used as well. Data segment descriptors may be similar to code segment descriptor 54, except that the D bit 42 is defined to indicate the upper bound of the segment or to define the default stack size (for stack segments).

5

Turning next to Fig. 4, a table 70 is shown illustrating the states of the LMA bit, the L bit in the code segment descriptor, and the D bit in the code segment descriptor and the corresponding operating mode of processor 10 according to one embodiment of processor 10. Other embodiments are possible and contemplated. As table 70 illustrates, if the LMA bit is clear, then the L bit is reserved (and defined to be zero). However, processor 10 may treat the L bit as a don't care if the LMA bit is clear. Thus, the x86 compatible 16 bit and 32 bit modes may be provided by processor 10 if the LMA bit is clear. If the LMA bit is set and the L bit in the code segment is clear, then a compatibility operating mode is established by processor 10 and the D bit selects 16 bit or 32 bit mode. If the LMA bit and the L bit are set and the D bit is clear, 32/64 mode is selected for processor 10. Finally, the mode which would be selected if the LMA, L and D bits are all set is reserved.

As mentioned above, the 32/64 operating mode includes a default address size in excess of 32 bits (implementation dependent but up to 64 bits) and a default operand size of 32 bits. The default operand size of 32 bits may be overridden to 64 bits via a particular instruction's encoding. The default operand size of 32 bits is selected to minimize average instruction length (since overriding to 64 bits involves including an instruction prefix in the instruction encoding which may increase the instruction length) for programs in which 32 bits are sufficient for many of the data manipulations performed by the program. For such programs (which may be a substantial number of the programs currently in existence), moving to a 64 bit operand size may actually reduce the execution performance achieved by the program (i.e. increased execution time). In part, this reduction may be attributable to the doubling in size in memory of the data structures

used by the program when 64 bit values are stored. If 32 bits is sufficient, these data structures would store 32 bit values, Thus, the number of bytes accessed when the data structure is accessed increases if 64 bit values are used where 32 bit values would be sufficient, and the increased memory bandwidth (and increased cache space occupied by each value) may cause increased execution time. Accordingly, 32 bits is selected as the default operand size and the default may be overridden via the encoding of a particular instruction.

Optional Segment Use and Unconditional Segment Load

Generally, the processor 10 is configured to selectively use segmentation based on which of the protected operating modes is active but to perform the segment load (including segment descriptor checks) no matter which of the protected operating modes is active. In this manner, segmentation may be effectively disabled in one or more protected operating modes, but code executing in that protected operating mode may still establish segments in the segment registers 24. Thus, for example, the operating system code may operate in the protected operating mode(s) in which segmentation is disabled, but the operating system may set up segments for application programs that use segmentation. Applications which do not use segmentation may execute in the protected mode(s) in which segmentation is disabled as well.

Turning next to Fig. 5, a flowchart is shown illustrating operation of one embodiment of the execution core 14 during instruction execution. Other embodiments are possible and contemplated. Only portions of the instruction execution related to segmentation are shown in Fig. 5. While the blocks shown are illustrated in a particular order for ease of understanding, any suitable order may be used. Additionally, blocks may be performed in parallel using combinatorial logic circuitry within the execution core 14 or in microcode included within execution core 14.

The execution core 14 determines if the active operating mode is the 32/64 mode

(decision block 80). In the illustrated embodiment, segmentation is at least partially disabled in 32/64 mode. More particularly, segment descriptor information from the segment register specified by the instruction is not used during execution of the instruction in 32/64 mode (with the exception of the FS and GS segment registers, in the illustrated embodiment). Thus, if the active operating mode is 32/64, execution core 14 determines if the segment register specified by the instruction is the FS or GS segment register (decision block 82). If the specified segment register is not FS or GS, the execution core 14 ignores the contents of the specified segment register (block 84). On the other hand, if the specified segment register is FS or GS, or if the active operating mode is not 32/64 mode, the specified segment register is accessed and the segment descriptor information therein is used as part of the instruction execution (block 86).

While the embodiment of Fig. 5 illustrates 32/64 mode as a protected operating mode in which segmentation is disabled for at least some segment registers, other embodiments may have any of one or more protected operating modes in which segmentation is disabled. For example, alternative 32 and/or 16 bit protected modes may be defined in which segmentation is disabled.

It is noted that block 84 may be implemented in a variety of fashions. For example, in one embodiment, a pseudo-segment register may be implemented in segment registers 24. The pseudo-segment register may have a segment base address of zero, a segment limit of the maximum size, and attributes allowing any type of access. The pseudo-segment register may be read instead of the segment register specified by the instruction if the active operating mode is one in which segmentation is disabled or partially disabled. Alternatively, a segment base address of zero, a segment limit of the maximum size, and the attributes may be muxed into the path from the segment registers to the logic which uses the segment information.

While the illustrated embodiment allows for the FS and GS segment registers to

be used while other data segment and/or stack segment registers are ignored in a protected mode in which segmentation is disabled, other embodiments may not use any segment registers, or any subset of the segment registers, as desired.

5 As used herein, an instruction "specifies a segment register" if the segment register is an operand of the instruction. The instruction may implicitly specify the segment register (e.g. as part of the definition of the instruction assigned to that particular opcode) or may explicitly specify the segment register (e.g. in an operand field or part of an operand value). In one implementation, a segment override prefix byte may be include
10 in the instruction to specify the segment register.

 The above description has referred to selectively using segmentation, or disabling segmentation (fully and/or partially). Generally speaking, the execution of an instruction specifying a given segment register may be "responsive to the segment descriptor"
15 indicated by the segment selector if information in the segment descriptor affects the execution. For example, the segment base address from the segment descriptor may be used in an address calculation during execution of the instruction. The generated address may be checked against the segment limit in the segment descriptor. Attributes from the segment descriptor may be used to limit the access to the segment during the execution of
20 the instruction. On the other hand, the execution of the instruction may not be responsive to the segment descriptor if information in the segment descriptor does not affect the execution. More particularly in one embodiment, the execution of the instruction may be selectively responsive to the segment descriptor dependent on the active protected operating mode (e.g. responsive if the active protected operating mode does not include
25 disabling or partially disabling segmentation and not responsive if the active protected operating mode does include disabling or partially disabling segmentation).

 Turning next to Fig. 6, a flowchart is shown illustrating operation of one embodiment of the execution core 14 during execution of a segment load instruction.

The flowchart of Fig. 6 may apply instead of the flowchart of Fig. 5 for the segment load instruction. Other embodiments are possible and contemplated. Only portions of the instruction execution related to segmentation are shown in Fig. 6. While the blocks shown are illustrated in a particular order for ease of understanding, any suitable order may be used. Additionally, blocks may be performed in parallel using combinatorial logic circuitry within the execution core 14 or may be performed in microcode included in execution core 14.

The segment load instruction specifies a segment selector as an operand and further specifies the segment register to be loaded with the segment selector. Additionally, as mentioned above, information from the segment descriptor located by the segment selector may be loaded into the segment register. Thus, the execution core 14 may read the indexed segment descriptor from the segment descriptor table (block 90). The execution core 14 may check the segment descriptor to ensure that the segment descriptor may be loaded into the specified segment register (block 92). Fig. 7 is a table illustrating checks performed during a segment load in the x86 architecture. The execution core 14 determines if an exception condition is detected during the check (decision block 94). If an exception condition is detected, the execution core 14 may cancel the segment load and exit (branch) to the exception vector for that exception condition (block 96). Different exception conditions detected in the segment check may have different exception vectors or the same exception vector, as desired. If an exception condition is not detected, the execution core 14 may update the specified segment register with the segment selector and segment descriptor information (block 98).

It is noted that the operating mode is not checked as part of the flowchart in Fig. 6. In other words, segment load instructions may execute the same way in any operating mode.

Turning now to Fig. 7, a table 100 illustrating exemplary segment descriptor

checks for one embodiment of the execution core 14 is shown. Other embodiments are possible and contemplated. Table 100 illustrates conditions which, if not met, result in an exception on the segment load instruction. Some of the conditions which are checked may depend on the segment register being loaded, while other conditions may be checked for any segment register load.

If the CS segment register is being loaded, the segment descriptor type (e.g. type field 52 in Figs. 2 and 3) may be checked to ensure that the segment is a code segment.

If the SS segment register is being loaded, the segment descriptor type may be checked to ensure that the segment is a writeable data segment. Additionally, the privilege level (e.g. DPL field 50 in Figs. 2 and 3 as well as a requested privilege level which may be part of the segment selector) may be checked to ensure that it matches the current privilege level (e.g. the privilege level in the CS segment register). Still further, the segment may be checked to ensure that it is not the null segment. The null segment may be a segment selector with an index of zero and a table indicator bit of zero.

If the DS, ES, FS, or GS segment register is being loaded, the segment descriptor type may be checked to ensure that the segment is a data segment or a readable code segment. Additionally, the privilege level may be checked to ensure that it is less than or equal to the current privilege level.

For any segment register, the selector index may be checked to ensure that it is within the descriptor table limit of the selected descriptor table, and that the segment is present (e.g. P bit 48 is set).

Turning next to Fig. 8, a flowchart is shown one embodiment of an operating system launching an application program. In the illustrated example, the operating system operates in a protected operating mode in which segmentation is not used (e.g.

32/64 mode) and the application program operates in a protected operating mode in which segmentation is used (e.g. 32 bit compatibility mode). Other embodiments are possible and contemplated. In the illustrated embodiment, the operating system portion is shown to the left of a vertical dotted line 110 and the application program portion is shown to the right of line 110.

The operating system creates the segment descriptors used by the application program, storing the created segment descriptors in the segment descriptor table (block 112). Additionally, one or more of the segment descriptors (and corresponding segment selectors locating the segment descriptors) may be loaded into segment registers using one or more segment load instructions (block 114). Since segment register load instructions operate in any operating mode, the segments may thus be established in the segment registers by the operating system for the application program. If one of the segment descriptors has an exception condition, processor 10 may exit to the exception handler (block 116), illustrated by a dashed line since the exit may not be part of the operating system code. After the exception is handled, control may return to the operating system routine illustrated in Fig. 8 or control may return to some other point. After successfully loading the desired segment descriptors into segment registers, the operating system routine may then branch to the application's code segment (block 118). The operating mode may then be changed to the application program's operating mode. The branch to the application's code segment may be performed through a call gate, or through a far branch, as desired.

The application program's code is executed (block 120), until a return to the operating system is detected (decision block 122). The return to the O/S may be due to an exception, an operating system call, or termination of the program, for example.

It is noted that, while Fig. 8 illustrates interaction between the operating system and an application program, a similar interaction may occur between any two code

sequences having different operating modes.

Software Embodiments

While the above description may generally have described a processor which may
5 directly support, in hardware, the processor architecture having the features described
above, it is contemplated that other processor embodiments may not directly implement
the processor architecture. Instead, such embodiments may directly implement a different
processor architecture (referred to below as a native processor architecture, which may
define a native instruction set including native instructions). Any native processor
10 architecture may be used. For example, the MIPS, Power PC, Alpha, Sparc, ARM, etc.
architectures may be used. The processor architecture may be implemented in software
executing on the native processor architecture in a variety of fashions, using any native
processor architecture such as, for example, the Crusoe products of Transmeta
Corporation.

15
Generally, a processor embodiment implementing a native processor architecture
different than the processor architecture described above (referred to below as the non-
native processor architecture) may support the non-native processor architecture in a
variety of fashions. For example, such a processor embodiment may execute interpreter
20 software which reads each non-native instruction in a non-native code sequence as data,
and executes various software routines which emulate the defined operation of the non-
native instruction as defined in the non-native processor architecture. Alternatively,
translator software may be executed. The translator software may translate the non-native
instructions in the code sequence to an equivalent set of native instructions defined by the
25 native instruction set architecture. The native code sequence may be stored in memory,
and may be executed instead of the corresponding non-native code sequence. In yet
another alternative, a mixture of interpretation and translation may be used. For example,
the code sequence may be interpreted, but the interpreter may also generate statistics

about which parts of the code sequence are being most frequently executed. The most frequently executed portions may then be translated to native code sequences.

In any of the above methods, the architected state defined by the non-native processor architecture may be maintained by the combination of the processor and the software (interpreter or translator) in a variety of fashions. For example, the non-native architected state may be mapped to memory locations in a memory addressable by the processor, to general registers defined by the native processor architecture (by software convention, either in the interpreter or in the translator), or the processor may directly support the non-native architected state by defining registers or other storage hardware within the processor that corresponds to the non-native architected state. The non-native architected state may be stored using any combination of the above methods, as desired.

Generally, the architected state includes any state defined to exist by the architecture. For example, in the above described embodiment, the non-native architected state may include general registers (e.g. RAX, RBX, etc.), segment registers, control registers, other registers such as the model specific registers (MSRs), etc. Additionally, the architected state may include data structures defined for the operating system to create, such as the descriptor tables, page tables, task state segments, etc.

Turning to Fig. 9, a flowchart illustrating an exemplary interpreter which may be used to interpret non-native instructions is shown. Other embodiments are possible and contemplated. While the blocks shown are illustrated in a particular order for ease of understanding, any suitable order may be used. Furthermore, blocks may be performed in parallel, as desired.

The blocks shown in Fig. 9 illustrate the emulation of one non-native instruction. Generally, the interpreter may execute the blocks shown in Fig. 9 for each non-native instruction to be executed according to the non-native code sequence to be executed.

The interpreter may determine the operating mode for the non-native instruction (block 1000). As described above, the operating mode may be determined from the LMA bit in control register 26 and the L bit and D bit from the code segment descriptor indicated by the CS segment register. The operating mode may be determined anew from the LMA, L bit, and D bit for each non-native instruction, or the resulting operating mode may be stored in a temporary register for access by the interpreter for each non-native instruction. If the resulting operating mode is stored, the interpreter may update the stored operating mode if an instruction modifies the CS segment register or interrupt or exception handling causes the operating mode to change. As mentioned above, the CS segment register and the control register(s) (which are part of the non-native architected state) may actually be memory locations, general registers, or special purpose registers, or any combination thereof.

The interpreter may read the current non-native instruction from memory, and may analyze the non-native instruction to determine the operations to be taken to emulate the non-native instruction (block 1002). The interpreter may read the non-native instruction one byte at a time, or may read a suitable set of consecutive bytes and process the bytes. For example, a native processor architecture in which operands are 32 bit may read 32 bits (4 bytes) of the non-native instruction at a time, and then may process the four bytes before reading any additional bytes.

Generally, the interpreter software may decode the non-native instruction in a manner analogous to processor 10 decoding the instruction in hardware. Thus, for the illustrated non-native processor architecture, which is compatible with the x86 processor architecture, the analyzing of the non-native instruction includes analyzing any prefix bytes which may precede the opcode byte, analyzing the opcode byte, analyzing the addressing mode (Mod R/M) byte (if present), and analyzing the scale-index-base (SIB) byte (if present). Prefix bytes may override the operating mode, and may also include

register specifier bits (e.g. the REX prefix byte). The opcode byte specifies the operation to be performed, and in some cases may include a register specifier or may implicitly specify an operand (e.g. the stack or the stack pointer). The Mod R/M byte specifies operands (including any displacement operands which may follow the Mod R/M byte or the SIB byte, if the SIB byte is present) and may include register specifiers. Finally, the SIB byte may include register specifiers. From the information gained from analyzing the non-native instruction, the interpreter has the information to emulate the non-native instruction (including operating mode for the non-native instruction, which specifies the operand size and address size of the non-native instruction, operands, the operation to be performed, etc.).

If the non-native instruction includes a memory operand (decision block 1004), the interpreter may calculate the effective address of the instruction (block 1006). If the non-native instruction has a memory operand, some of the operands identified in block 1002 may be address operands used to generate the effective address. Thus, the interpreter may read the address operands from the non-native architected state and may add them to generate an effective address. The size of the effective address may be determined by the address size for the instruction, as determined at blocks 1000 and 1002. It is noted that the native processor architecture may support an address size which is less than the address size supported by the non-native processor architecture. For example, in one exemplary embodiment described above, the virtual address size may be 48 bits in 32/64 mode. The native processor may, for example, support a virtual address size of 32 bits. In such an embodiment, block 1006 may represent a series of calculations in which the least significant bits (e.g. 32 bits) of the virtual address may be calculated, and any carry from the least significant bits may be carried into a calculation of the most significant bits of the virtual address.

The interpreter may then perform the operation specified by the non-native instruction (block 1008). If the non-native instruction includes a memory operand as a

source operand, the interpreter may read the memory operand from the effective address calculated at block 1006. Other operands may be read from the non-native architected state. The operation may include an arithmetic operation, a logical operation, a shift, a move to another storage location, etc. The native processor architecture may support an operand size smaller than the operand size of the instruction. In such cases, performing the operation may include multiple calculations on portions of the operand to calculate the result.

The interpreter determines if the non-native instruction resulted in an exception (decision block 1010). Generally, exceptions may occur throughout the execution of the operations specified by the non-native instruction. For example, accessing a source memory operand may result in a page fault before any of the actual instruction operation is performed. During the operations, various architecturally-defined exceptions may also occur. The interpreter may interrupt processing of the non-native instruction upon detecting an exception, and may branch to exception handler instructions (block 1012). The exception handler may be native code or non-native code or a combination thereof, as desired. If the non-native processor architecture specifies the update of any architected state when an exception is taken (e.g. various control registers may store the address of the exception causing instruction, the exception reason, etc.), the interpreter may update the non-native architected state as defined.

It is noted that the interpreter software is executing on the native processor, and thus is subject to experiencing exceptions as defined in the native processor architecture. These exceptions may generally be different from the exceptions detected by the interpreter software, which are exceptions experienced by the non-native code being interpreted according to the non-native processor architecture.

If no exception occurs during emulation of the non-native instruction, the interpreter may update the non-native architected state according to the definition of the

non-native instruction (block 1014). Finally, the interpreter may calculate the next non-native instruction fetch address to fetch the next instruction (block 1016). The next fetch address may be sequential to the current non-native instruction, or may be a different address (e.g. if the current non-native instruction is a taken branch, the next fetch address may be the target address of the branch instruction).

It is noted that the interpreter may operate in protected mode, using virtual addresses. In other words, the effective address calculated at block 1006 may be a virtual address which is translated by the translation mechanism specified by the non-native processor architecture to a physical address. The processor may include a translation lookaside buffer (TLB) used to cache translations. The processor may either support reload of the TLB from the non-native translation tables (page tables), or may take an exception on a TLB miss to allow software reload of the TLB.

Generally, the interpreter may perform the flowcharts of Figs. 5 and 6 at any suitable point in the processing of instructions, e.g. blocks 1000, 1002, 1006 or 1008, depending on the instruction.

Turning to Fig. 10, a flowchart illustrating an exemplary translator which may be used to translate non-native instructions in the non-native processor architecture to native instructions in the native processor architecture. Other embodiments are possible and contemplated. While the blocks shown are illustrated in a particular order for ease of understanding, any suitable order may be used. Furthermore, blocks may be performed in parallel, as desired.

The blocks shown in Fig. 10 illustrate the translation of one non-native code sequence responsive to a fetch address for the first instruction in the non-native code sequence. The code translator may translate any number of non-native instructions to produce a translated code sequence having native instructions. For example, the

translator may translate from the initial non-native instruction to a basic block boundary (i.e. a branch instruction). Alternatively, the translator may speculatively translate two or more basic blocks or may translate up to a maximum number of non-native or resulting native instructions, if desired.

5

Generally, the translator may maintain a translation cache which stores translated code sequences previously produced by the translator. The translation cache may identify translated code sequences by the fetch address of the first non-native instruction in the corresponding non-native code sequences. Thus, the translator may determine if a translated code sequence corresponding to the fetch address is stored in the translation cache (decision block 1030). If there is a translated code sequence in the translation cache, the translator may cause the processor to branch to that translated code sequence (block 1032). On the other hand, if there is no translated code sequence, the translator may translate one or more non-native instructions from the non-native code sequence into native instructions in a translated code sequence (block 1034).

Generally, the translator may translate each non-native instruction into one or more native instructions which, when executed, may perform the same operation on the non-native architected state that the non-native instruction would have performed. The translator may generally perform the same decoding of instructions as is performed by the interpreter (block 1002 in Fig. 9) to determine what operations may need to be performed. For example, if the native processor architecture is a load/store architecture in which memory operands are accessed using explicit load/store instructions and other instruction use only register operands, load and store instructions may be used to access the memory operands and other instructions may be used to perform the explicit operation of a non-native instruction having a memory operand. The translated instructions may make use of temporary registers to hold intermediate values corresponding to the execution of the non-native instruction. Additionally, the translated instructions may access the non-native architected state to retrieve operands and may update the non-native architected

state with the final results of the non-native instruction. Generally, the native instructions corresponding to the non-native instruction may perform all of the operations defined for the instruction (e.g. blocks 1006, 1008, 1010, 1014, and 1016 in Fig. 9).

5 Once the translator has determined to terminate translation and save the translated sequence for execution, the translator may optionally optimize the translated code sequence (block 1036). The optimizations may include reordering the translated instructions for quicker execution, eliminating redundancies (e.g. redundant memory references, which may occur if multiple non-native instructions in the source code
10 sequence accessed the same memory location), etc. Any suitable set of optimizations may be used. The resulting translated code sequence may then be stored into the translation cache. Additionally, the processor may branch to the translated code sequence and execute the sequence (block 1032).

15 It is noted that, while the above description may refer to accessing and/or updating non-native architected state, including various registers, the non-native architected state may be stored in any suitable fashion. For example, architected registers may actually be stored in memory locations, as highlighted above. The mapping of architected registers from the non-native processor architecture to memory locations may be used in either of
20 the interpreter or the translator embodiments, or combinations thereof, to locate the non-architected state used during execution of the non-native instruction or affected by the execution of the non-native instruction. Thus, instructions which access the non-native architected state may perform memory reads/writes or register reads/writes, as the case may be.

25

Turning next to Fig. 11, a block diagram illustrating one exemplary mapping of non-native architected state to either memory locations in a memory 1040 or to processor resources in a native processor 1042. Native processor 1042 includes a register file 1044

including the architected general registers of the native processor architecture. Any number of registers may be provided.

5 In the embodiment of Fig. 11, all of the non-native architected state is mapped to memory 1040. For example, descriptor tables 1046 (which may include a global descriptor table, a local descriptor table, and an interrupt descriptor table), page tables 1048 (which store virtual to physical address translations), task state segments 1050, general registers 1052, segment registers 1054, control registers 1056, and other registers 1058 may represent non-native architected state.

10

Thus, in the embodiment of Fig. 11, to access any non-native architected state, a memory access may be performed. For example, if a non-native instruction has one of the general registers as an operand, the interpreter or translated native instruction performs a memory access to the memory location mapped to that general register to
15 access or update that general register. The registers in register file 1044 may be used by the interpreter or translator as temporary registers to hold intermediate results or for other local interpreter/translator state.

General registers 1052 may include integer general registers (e.g. RAX, RBX, etc.
20 as described above), the additional integer general registers defined by the REX prefix byte, floating point registers, Streaming Single Instruction, Multiple Data (SIMD) Extension (SSE) registers, and the additional SSE registers defined by the REX prefix byte.

25 Segment registers 1054 may include storage locations corresponding to the segment registers 24 shown in Fig. 1 above.

Control registers 1056 may include storage locations corresponding to various control registers defined in the non-native processor architecture. For example, control

registers storing the LMA, LME, PG and PE bits, as well as the LDTR and GDTR registers and the CR3 register (which stores the base address of the page tables 1048) are shown. Other control registers may be included as well.

5 Other registers 1058 includes any remaining architected registers. For example, the EFLAGS register (which stores condition code information), the instruction pointer (RIP) register (which stores the address of the instruction to be executed), and the model specific registers (MSRs) may be included in other registers 1058.

10 While the example of Fig. 11 maps all of the non-native architected state to memory 1040, other embodiments may implement other mappings. In Fig. 12, for example, some of the general registers in register file 1044 are mapped to the general registers 1052. Accordingly, if a non-native instruction has a general register as an operand, the interpreter accesses the corresponding register in register file 1044.

15 Similarly, the translator generates a translated instruction having the corresponding register in register file 1044 as an operand. Other architected state may still be accessed via memory operations in the embodiment of Fig. 12. Other registers in register file 1044 which are not assigned to non-native architected state may again be used as temporary registers for interpreter or translator use, as described above.

20 While the embodiment of Fig. 12 illustrates mapping the general registers 1052 to registers in register file 1044, any other non-native architected state may be mapped to registers in register file 1044. For example, any of segment registers 1054, control registers 1056, or other registers 1058 (or portions of any of these registers) may be

25 mapped to register file 1044, as desired.

Fig. 13 illustrates another example in which the general registers 1052 and the EFLAGS and RIP registers are mapped to registers in register file 1044. Additionally, in the example of Fig. 13, the segment registers 1054 are implemented in hardware in

processor 1042. More specifically, processor 1042 may not only implement storage for segment registers 1054, but may also include logic to generate the operating mode for instructions based on the information in the segment registers. Furthermore, for compatibility modes, the logic may include limit checks and attribute checks to ensure that accesses to the segment attempted by the non-native instructions (or the non-native instructions in the interpreter or the translated code sequence which correspond to the non-native instructions) are permitted.

Similarly, other embodiments may implement various control registers 1056 or other registers 1058 in hardware, including corresponding logic to act on the contents of the registers as defined in the non-native architecture. Generally, various embodiments of processor 1042 may implement any non-native architected state in hardware. Certain architected state may generally be implemented in memory since the non-native processor architecture defines the state to be in memory (e.g. descriptor tables 1046, pages tables 1048, and task state segments 1050). Such memory-based architected state may be cached in caches within processor 1042 (e.g. TLBs for page table information, hidden segment register portions for segment descriptor information, etc.).

As the above discussion illustrates, the non-native architected state may be stored in any suitable storage location. Generally, a storage location is a location capable of storing a value. Suitable storage locations may include, in various embodiments, a memory location, a general register mapped to the non-native architected state, or a special purpose register (which may include additional hardware to interpret the contents of the register), depending upon the embodiment. Additionally, suitable storage locations could include a scratch pad RAM (such as a portion of a cache predetermined to be used as scratch pad RAM).

Fig. 14 is a block diagram of one embodiment of a carrier medium 1090. Other embodiments are possible and contemplated. In the embodiment of Fig. 14, carrier medium 1090 stores an interpreter program 1092 and a translator program 1094.

5 Generally speaking, a carrier medium may include storage media such as magnetic or optical media, e.g., disk or CD-ROM, volatile or non-volatile memory media such as RAM (e.g. SDRAM, RDRAM, SRAM, etc.), ROM, etc., as well as transmission media or signals such as electrical, electromagnetic, or digital signals, conveyed via a communication medium such as a network and/or a wireless link. Carrier medium 1090
10 may thus be coupled to a computer system including processor 1042, may be part of a computer system including processor 1042, or may be a communication medium on which the computer system is capable of communicating. Computer systems including processor 1042 may be of any construction. For example, computer systems similar to those shown in Figs. 15 and 16 may be suitable.

15 Interpreter program 1090 may operate according to the flowchart of Fig. 9. Translator program 1094 may operate according to the flowchart of Fig. 10. Generally, interpreter program 1092 and translator program 1094 may each comprise code sequences including native instructions.

20

Computer Systems

Turning now to Fig. 15, a block diagram of one embodiment of a computer system 200 including processor 10 coupled to a variety of system components through a bus bridge 202 is shown. Other embodiments are possible and contemplated. In the
25 depicted system, a main memory 204 is coupled to bus bridge 202 through a memory bus 206, and a graphics controller 208 is coupled to bus bridge 202 through an AGP bus 210. Finally, a plurality of PCI devices 212A-212B are coupled to bus bridge 202 through a PCI bus 214. A secondary bus bridge 216 may further be provided to accommodate an electrical interface to one or more EISA or ISA devices 218 through an EISA/ISA bus

220. Processor 10 is coupled to bus bridge 202 through a CPU bus 224 and to an optional L2 cache 228. Together, CPU bus 224 and the interface to L2 cache 228 may comprise an external interface to which external interface unit 18 may couple.

5 Bus bridge 202 provides an interface between processor 10, main memory 204, graphics controller 208, and devices attached to PCI bus 214. When an operation is received from one of the devices connected to bus bridge 202, bus bridge 202 identifies the target of the operation (e.g. a particular device or, in the case of PCI bus 214, that the target is on PCI bus 214). Bus bridge 202 routes the operation to the targeted device.
10 Bus bridge 202 generally translates an operation from the protocol used by the source device or bus to the protocol used by the target device or bus.

 In addition to providing an interface to an ISA/EISA bus for PCI bus 214, secondary bus bridge 216 may further incorporate additional functionality, as desired. An
15 input/output controller (not shown), either external from or integrated with secondary bus bridge 216, may also be included within computer system 200 to provide operational support for a keyboard and mouse 222 and for various serial and parallel ports, as desired. An external cache unit (not shown) may further be coupled to CPU bus 224 between
20 processor 10 and bus bridge 202 in other embodiments. Alternatively, the external cache may be coupled to bus bridge 202 and cache control logic for the external cache may be integrated into bus bridge 202. L2 cache 228 is further shown in a backside configuration to processor 10. It is noted that L2 cache 228 may be separate from processor 10, integrated into a cartridge (e.g. slot 1 or slot A) with processor 10, or even integrated onto a semiconductor substrate with processor 10.

25

 Main memory 204 is a memory in which application programs are stored and from which processor 10 primarily executes. A suitable main memory 204 comprises DRAM (Dynamic Random Access Memory). For example, a plurality of banks of SDRAM (Synchronous DRAM) or Rambus DRAM (RDRAM) may be suitable.

PCI devices 212A-212B are illustrative of a variety of peripheral devices. The peripheral devices may include devices for communicating with another computer system to which the devices may be coupled (e.g. network interface cards, modems, etc.).

5 Additionally, peripheral devices may include other devices, such as, for example, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards. Similarly, ISA device 218 is illustrative of various types of peripheral devices, such as a modem, a sound card, and a variety of data acquisition cards such as GPIB or field bus interface cards.

10

Graphics controller 208 is provided to control the rendering of text and images on a display 226. Graphics controller 208 may embody a typical graphics accelerator generally known in the art to render three-dimensional data structures which can be effectively shifted into and from main memory 204. Graphics controller 208 may
15 therefore be a master of AGP bus 210 in that it can request and receive access to a target interface within bus bridge 202 to thereby obtain access to main memory 204. A dedicated graphics bus accommodates rapid retrieval of data from main memory 204. For certain operations, graphics controller 208 may further be configured to generate PCI protocol transactions on AGP bus 210. The AGP interface of bus bridge 202 may thus
20 include functionality to support both AGP protocol transactions as well as PCI protocol target and initiator transactions. Display 226 is any electronic display upon which an image or text can be presented. A suitable display 226 includes a cathode ray tube ("CRT"), a liquid crystal display ("LCD"), etc.

25 It is noted that, while the AGP, PCI, and ISA or EISA buses have been used as examples in the above description, any bus architectures may be substituted as desired. It is further noted that computer system 200 may be a multiprocessing computer system including additional processors (e.g. processor 10a shown as an optional component of computer system 200). Processor 10a may be similar to processor 10. More particularly,

processor 10a may be an identical copy of processor 10. Processor 10a may be connected to bus bridge 202 via an independent bus (as shown in Fig. 15) or may share CPU bus 224 with processor 10. Furthermore, processor 10a may be coupled to an optional L2 cache 228a similar to L2 cache 228.

5

Turning now to Fig. 16, another embodiment of a computer system 300 is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 16, computer system 300 includes several processing nodes 312A, 312B, 312C, and 312D. Each processing node is coupled to a respective memory 314A-314D via a memory controller 316A-316D included within each respective processing node 312A-312D. Additionally, processing nodes 312A-312D include interface logic used to communicate between the processing nodes 312A-312D. For example, processing node 312A includes interface logic 318A for communicating with processing node 312B, interface logic 318B for communicating with processing node 312C, and a third interface logic 318C for communicating with yet another processing node (not shown). Similarly, processing node 312B includes interface logic 318D, 318E, and 318F; processing node 312C includes interface logic 318G, 318H, and 318I; and processing node 312D includes interface logic 318J, 318K, and 318L. Processing node 312D is coupled to communicate with a plurality of input/output devices (e.g. devices 320A-320B in a daisy chain configuration) via interface logic 318L. Other processing nodes may communicate with other I/O devices in a similar fashion.

20

Processing nodes 312A-312D implement a packet-based link for inter-processing node communication. In the present embodiment, the link is implemented as sets of unidirectional lines (e.g. lines 324A are used to transmit packets from processing node 312A to processing node 312B and lines 324B are used to transmit packets from processing node 312B to processing node 312A). Other sets of lines 324C-324H are used to transmit packets between other processing nodes as illustrated in Fig. 16. Generally, each set of lines 324 may include one or more data lines, one or more clock lines corresponding to the data lines, and one or more control lines indicating the type of

25

30

packet being conveyed. The link may be operated in a cache coherent fashion for communication between processing nodes or in a noncoherent fashion for communication between a processing node and an I/O device (or a bus bridge to an I/O bus of conventional construction such as the PCI bus or ISA bus). Furthermore, the link may be operated in a non-coherent fashion using a daisy-chain structure between I/O devices as shown. It is noted that a packet to be transmitted from one processing node to another may pass through one or more intermediate nodes. For example, a packet transmitted by processing node 312A to processing node 312D may pass through either processing node 312B or processing node 312C as shown in Fig. 16. Any suitable routing algorithm may be used. Other embodiments of computer system 300 may include more or fewer processing nodes than the embodiment shown in Fig. 16.

Generally, the packets may be transmitted as one or more bit times on the lines 324 between nodes. A bit time may be the rising or falling edge of the clock signal on the corresponding clock lines. The packets may include command packets for initiating transactions, probe packets for maintaining cache coherency, and response packets from responding to probes and commands.

Processing nodes 312A-312D, in addition to a memory controller and interface logic, may include one or more processors. Broadly speaking, a processing node comprises at least one processor and may optionally include a memory controller for communicating with a memory and other logic as desired. More particularly, each processing node 312A-312D may comprise one or more copies of processor 10. External interface unit 18 may include the interface logic 318 within the node, as well as the memory controller 316.

Memories 314A-314D may comprise any suitable memory devices. For example, a memory 314A-314D may comprise one or more RAMBUS DRAMs (RDRAMs), synchronous DRAMs (SDRAMs), static RAM, etc. The address space of computer

system 300 is divided among memories 314A-314D. Each processing node 312A-312D may include a memory map used to determine which addresses are mapped to which memories 314A-314D, and hence to which processing node 312A-312D a memory request for a particular address should be routed. In one embodiment, the coherency point for an address within computer system 300 is the memory controller 316A-316D coupled to the memory storing bytes corresponding to the address. In other words, the memory controller 316A-316D is responsible for ensuring that each memory access to the corresponding memory 314A-314D occurs in a cache coherent fashion. Memory controllers 316A-316D may comprise control circuitry for interfacing to memories 314A-314D. Additionally, memory controllers 316A-316D may include request queues for queuing memory requests.

Generally, interface logic 318A-318L may comprise a variety of buffers for receiving packets from the link and for buffering packets to be transmitted upon the link. Computer system 300 may employ any suitable flow control mechanism for transmitting packets. For example, in one embodiment, each interface logic 318 stores a count of the number of each type of buffer within the receiver at the other end of the link to which that interface logic is connected. The interface logic does not transmit a packet unless the receiving interface logic has a free buffer to store the packet. As a receiving buffer is freed by routing a packet onward, the receiving interface logic transmits a message to the sending interface logic to indicate that the buffer has been freed. Such a mechanism may be referred to as a "coupon-based" system.

I/O devices 320A-320B may be any suitable I/O devices. For example, I/O devices 320A-320B may include devices for communicate with another computer system to which the devices may be coupled (e.g. network interface cards or modems). Furthermore, I/O devices 320A-320B may include video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards, sound cards, and a variety of data acquisition cards such as

GPIB or field bus interface cards. It is noted that the term "I/O device" and the term "peripheral device" are intended to be synonymous herein.

Numerous variations and modifications will become apparent to those skilled in
5 the art once the above disclosure is fully appreciated. It is intended that the following
claims be interpreted to embrace all such variations and modifications.